Extension de classes

Alexis Nasr (d'après les slides de Arnaud Labourel)

Extension

L'extension permet de créer une classe en :

- conservant les services (attributs et méthodes) d'une autre classe;
- ajoutant de nouveaux services (attributs et méthodes);
- redéfinissant certains services (méthodes).

En Java :

- On utilise le mot-clé `extends` pour étendre une classe ;
- Une classe ne peut étendre qu'une seule classe.

Terminologie

- Si la classe `B` étend la classe `A`, on dit que `A` est la **super classe** de `B`.
- On dit aussi que `B` **hérite** de la classe `A`.

Extension pour ajouter des nouveaux services

Supposons que nous ayons la classe Point suivante :

```
public class Point {
  public int x, y;
  public void setPosition(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

Il est possible d'ajouter de nouveaux services en utilisant l'extension :

```
public class Pixel extends Point {
  public int r, g, b;
  public void setColor(int r, int g, int b) {
    this.r = r; this.g = g; this.b = b; }
}
```

Extension pour ajouter des nouveaux services

Les méthodes et attributs de `Point` sont disponibles dans `Pixel` :

```
Pixel pixel = new Pixel();
pixel.setPosition(4,8);
System.out.println(pixel.x); // → 4
System.out.println(pixel.y); // → 8
pixel.setColor(200,200,120);
```

Évidemment, ceux de `Pixel` ne sont pas disponibles dans `Point` :

```
Point point = new Point();
point.setPosition(4,8);
System.out.println(point.x); // → 4
System.out.println(point.y); // → 8
point.setColor(200,200,120); // impossible !
```

Redéfinition de méthode

On ajoute la méthode `clear()` à la classe `Point` :

```
public class Point {
  public int x, y;
  public void setPosition(int x, int y) {
    this.x = x; this.y = y;
  }
  public void clear() {
    x = 0; y = 0;
  }
}
```

Redéfinition de méthode

Il est possible de redéfinir la méthode `clear` dans `Pixel` :

```
public class Pixel extends Point {
  public int r, g, b;
  public void setColor(int r, int g, int b) {
    this.r = r; this.g = g; this.b = b;
    }
  public void clear(){
    x = 0; y = 0;
    r = 0; g = 0; b = 0;
  }
}
```

L'annotation @Override

Il est possible d'indiquer explicitement qu'une méthode redéfinit une méthode héritée à l'aide de l'annotation `@Override`

```
public class Pixel extends Point {
  public int r, g, b;
  public void setColor(int r, int g, int b) {
    this.r = r; this.g = g; this.b = b;
    }
  @Override
  public void clear(){
    x = 0; y = 0;
    r = 0; g = 0; b = 0;
  }
}
```

L'annotation @Override

- Elle permet d'indiquer qu'une méthode redéfinit une méthode héritée
- Elle n'est pas obligatoire
- Elle permet de rendre le code plus explicite
- Elle permet, lors de la compilation, de vérifier que le programmeur a bien respecté la signature de la méthode héritée

```
public class Pixel extends Point {
  public int r, g, b;
  public void setColor(int r, int g, int b) {
    this.r = r; this.g = g; this.b = b;
    }
  @Override
  public void claer(){ // erreur détectée par le compilateur
                       // elle ne l'aurait pas été sans @Override
    x = 0; y = 0;
    r = 0; g = 0; b = 0;
  }
}
```

Redéfinition avec super

```
public class Point {
  // ...
  public void clear() {
    setPosition(0, 0);
  }
}
```

Le mot-clé `super` permet d'utiliser la méthode `clear` de Point :

```
public class Pixel extends Point {
  // ...
  public void clear() {
    super.clear();
    setColor(0, 0, 0);
    }
}
```

Le mot-clé super

```
public class Point {
  public int x, y;
  public void setPosition(int x, int y) {
    this.x = x; this.y = y; }
}
```

Si la méthode n'a pas été redéfinie, le mot clé super est inutile :

```
public class Pixel extends Point {
  public int r, g, b;
  public void setColor(int r, int g, int b) {
    this.r = r; this.g = g; this.b = b;
  }
  public void clear() {
    /*super.*/setPosition(0, 0);
    setColor(0, 0, 0);
  }
}
```

Les constructeurs et le mot-clé super

```
public class Point {
  public int x, y;
  public Point(int x, int y) {
    this.x = x; this.y = y;
  }
}
```

La classe Point n'a pas de constructeur sans paramètre, il faut donc indiquer comment initialiser la partie de Pixel issue de la classe Point :

```
public class Pixel extends Point {
  public int r, g, b;
  public Pixel(int x, int y, int r, int g, int b) {
    super(x, y); // appel du constructeur de Point
    this.r = r; this.g = g; this.b = b;
  }
}
```

Les constructeurs

Supposons que nous ayons la classe `Point` suivante :

```
public class Point {
  public int x, y;
  public Point() { x = 0; y = 0; }
  public Point(int x, int y) { this.x = x; this.y = y; }
}
```

Par défaut, le constructeur sans paramètre est appelé :

```
public class Pixel extends Point {
  public int r, g, b;
  public Pixel(int r, int g, int b) {
    // appel du constructeur sans paramètre de Point
    this.r = r; this.g = g; this.b = b;
  }
}
```

Les constructeurs

```
public class Point {
  public int x, y;
  //public Point() { x = 0; y = 0; }
  public Point(int x, int y) { this.x = x; this.y = y; }
}
```

Ici, vous devez préciser les paramètres du constructeur avec `super` :

```
public class Pixel extends Point {
  public int r, g, b;
  public Pixel(int r, int g, int b) {
  // erreur de compilation
  // (aucun constructeur sans paramètre)
  this.r = r; this.g = g; this.b = b;
  }
}
```

Transtypages et polymorphisme

Aucune méthode ou attribut ne peut être supprimé lors d'une extension. Par exemple, `Pixel` possède toutes les attributs et méthodes de `Point` (même si certaines méthodes ont pu être redéfinies).

Par conséquent, l'**upcasting** est toujours autorisé :

```
Point point = new Pixel();
point.setPosition(2,4);
System.out.println(point.x + " " + point.y);
point.clear();
```

Remarques

- Le code exécuté lors d'un appel de méthode est déterminé à l'exécution en fonction de la référence présente dans la variable.
- Le typage des variables permet de vérifier à la compilation l'existence des attributs et des méthodes.

Upcasting

Définition d'upcasting

Considérer une instance d'une classe comme une instance d'une de ses **super-classes**, c'est-à-dire :

- une classe étendue par la classe
- une interface implémentée par la classe

Remarques

- Après l'*upcasting*, les services supplémentaires de la classe (méthodes et attributs non-disponibles dans la super-classe) ne sont plus accessibles.
- Lors de l'appel des méthodes, si la classe a redéfini (*overrides*) la méthode, c'est le code de la classe qui est exécuté.

La classe Object

Par défaut, les classes étendent la classe `Object` de Java. Par conséquent, l'*upcasting* vers la classe `Object` est toujours possible :

```
Pixel pixel = new Pixel();
Object object = pixel; // upcasting
Object[] array = new Object[10];
for (int i = 0; i < t; i++) {
  if (i%2==0) array[i] = new Point(); // upcasting
  else array[i] = new Pixel(); // upcasting
}
```

Note

`object.setPosition(2,3)` n'est pas autorisé dans le code ci-dessus car la classe `Object` ne possède pas la méthode `setPosition` et seul le type de la variable compte pour déterminer si l'appel d'une méthode ou l'utilisation d'un attribut est autorisé.

Quelques méthodes de la classe `Object`

- `protected Object clone()`: Creates and returns a copy of this object.
- `boolean equals(Object obj)`: Indicates whether some other object is "equal to" this one.
- `Class<?> getClass()`: Returns the runtime class of this Object.
- `int hashCode()`: Returns a hash code value for the object.
- `String toString()`: Returns a string representation of the object.

La méthode `toString()` de la classe Object

Par transitivité de l'extension, toutes les méthodes et attributs de la classe `Object` sont disponibles sur toutes les instances :

```
Object object = new Object();
Point point = new Point(2,3);
System.out.println(object.toString());
// → java.lang.Object@19189e1
System.out.println(point.toString());
// → test.Point@7c6768
```

Le résultat est un peu décevant !

La méthode `toString()` et le polymorphisme (1/2)

Évidemment, il est possible de redéfinir la méthode `toString` :

```
public class Point {
  public int x, y;
  public Point(int x, int y) {
    this.x = x; this.y = y;
  }

  public String toString() {
    return "("+x+","+y+")";
  }
}
```

La méthode `toString()` et le polymorphisme (2/2)

Le polymorphisme fait que cette méthode est appelée si la variable contient une référence vers un `Point` :

```
Point point = new Point(2,3);
Object object = point;
System.out.println(point); // → (2,3)
System.out.println(object); // → (2,3)
```